Dynamic Local Searchable Symmetric Encryption

Brice Minaud* and Michael Reichle*

*DIENS, École normale supérieure, PSL University, CNRS, INRIA, 75005 Paris, France.

Abstract. In this article, we tackle for the first time the problem of *dynamic* memory-efficient Searchable Symmetric Encryption (SSE). In the term "memory-efficient" SSE, we encompass both the goals of *local* SSE, and *page-efficient* SSE. The centerpiece of our approach is a novel connection between those two goals. We introduce a map, called the Generic Local Transform, which takes as input a *page-efficient* SSE scheme with certain special features, and outputs an SSE scheme with strong *locality* properties. We obtain several results. (1) First, for page-efficient SSE with page size p, we build a *dynamic* scheme with storage efficiency $\mathcal{O}(1)$ and page efficiency $\widetilde{\mathcal{O}}(\log\log(N/p))$, called LayeredSSE. The main technical innovation behind LayeredSSE is a novel weighted extension of the two-choice allocation process, of independent interest. (2) Second, we introduce the Generic Local Transform, and combine it with LayeredSSE to build a *dynamic* SSE scheme with storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\widetilde{\mathcal{O}}(\log\log N)$, under the condition that the longest list is of size $\mathcal{O}\left(N^{1-1/\log\log\lambda}\right)$. This matches, in every respect, the purely *static* construction of Asharov et al. presented at STOC 2016: dynamism comes at no extra cost. (3) Finally, by applying the Generic Local Transform to a variant of the Tethys scheme by Bossuat et al. from Crypto 2021, we build an unconditional static SSE with storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\mathcal{O}(\log^{\varepsilon} N)$, for an arbitrarily small constant $\varepsilon > 0$. To our knowledge, this is the construction that comes closest to the lower bound presented by Cash and Tessaro at Eurocrypt 2014.

1 Introduction

Searchable Symmetric Encryption. In Searchable Symmetric Encryption (SSE), a client outsources the storage of a set of documents to an untrusted server. The client wishes to retain the ability to search the documents, by issuing search queries to the server. In the setting of *dynamic* SSE, the client may also issue update queries, in order to modify the contents of the database, for instance by adding or removing entries. The server must be able to correctly process all queries, while learning as little information as possible about the client's data and queries. SSE is relevant in many cloud storage scenarios: for example, in cases such as outsourcing the storage of a sensitive database, or offering an encrypted messaging service, some form of search functionality may be highly desirable.

In theory, SSE is a special case of computation on encrypted data, and could be realized using generic solutions, such as Fully Homomorphic Encryption. In practice, such approaches incur a large performance penalty. Instead, SSE schemes typically aim for high-performance solutions, scalable to large real-world databases. Towards that end, SSE trades off security for efficiency. The server is allowed to learn some information about the client's data. For example, SSE schemes typically leak to the server the repetition of queries (*search pattern*), and the identifiers of the documents that match a query (*access pattern*). The security model of SSE is parametrized by a *leakage function*, which specifies the nature of the information leaked to the server.

Locality. In the case of single-keyword SSE, search queries ask for all documents that contain a given keyword. To realize that functionality, the server maintains an (encrypted) reverse index, where each keyword is mapped to the list of identifiers of documents that match the keyword. When the client wishes to search for the documents that match a given keyword, the client simply retrieves the corresponding list from the server. A subtle issue, however, is how the lists should be stored and accessed by the server.

The naive approach of storing one list after the other is unsatisfactory: indeed, the position of a given list in memory becomes dependent on the lengths of other lists, thereby leaking information about those lists. A common approach to address that issue is to store each list element at a random location in memory. In that case, when retrieving a list, the server must visit as many random memory locations as the number of elements in the list. This is also undesirable, for a different reason: for virtually all modern storage media, accessing many random memory locations is much more expensive than visiting one continuous region. Because SSE relies on fast symmetric cryptographic primitives, the cost of memory accesses becomes the performance bottleneck. To capture that cost, [CT14] introduces the notion of *locality*: in short, the locality of an SSE scheme is the number of discontinuous memory locations that the server must access to answer a query.

The two extreme solutions outlined above suggest a conflict between security and locality. At Eurocrypt 2014, Cash and Tessaro showed that this conflict is inherent [CT14]: if a secure SSE scheme has constant storage efficiency (the size of the encrypted database is linear in the size of the plaintext database), and constant read efficiency (the amount of data read by the server to answer a search query is linear in the size of the plaintext answer), then it cannot have constant locality.

Local SSE constructions. Since then, many SSE schemes with constant locality have been proposed, typically at the cost of superconstant read efficiency. At STOC 2016, Asharov et al. presented a scheme with $\mathcal{O}(1)$ storage efficiency, $\mathcal{O}(1)$ locality, and $\widetilde{\mathcal{O}}(\log N)$ read efficiency, where N is the size of the database [ANSS16]. At Crypto 2018, Demertzis et al. improved the read efficiency to $\mathcal{O}\left(\log^{2/3+\varepsilon} N\right)$ [DPP18]. Several trade-offs with $\omega(1)$ storage efficiency were also proposed in [DP17]. When the size of the longest list in the database is bounded, stronger results are known. When such an upper bound is required, we

will call the construction *conditional*. The first conditional SSE is due to Asharov et al., and achieves $\widetilde{\mathcal{O}}(\log\log N)$ read efficiency, on the condition that the size of the longest list is $\mathcal{O}\big(N^{1-1/\log\log N}\big)$. This was later improved to $\widetilde{\mathcal{O}}(\log\log\log N)$ read efficiency, with a stronger condition of $\mathcal{O}\big(N^{1-1/\log\log\log N}\big)$ on the size of the longest list.

Locality was introduced as a performance measure for memory accesses, assuming an implementation on Hard Disk Drives (HDDs). In [BBF+21], Bossuat et al. show that in the case of Solid State Drives (SSDs), such as flash disks, locality is no longer the relevant target. Instead, performance is mainly determined by the number of memory pages accessed, regardless of whether they are contiguous. In that setting the right performance metric is *page efficiency*. Page efficiency is defined as the number of pages read by the server to answer a query, divided by the number of pages needed to store the plaintext answer. The main construction of [BBF+21] achieves $\mathcal{O}(1)$ storage efficiency and $\mathcal{O}(1)$ page efficiency, assuming a client-side memory of $\omega(\log\lambda)$ pages.

To this day, a common point among all existing constructions, both local and page-efficient, is that they are purely *static*, as known techniques for sublogarithmic read efficiency and page efficiency do not apply to the dynamic setting. That may be because of the difficulty inherent in building local SSE, even in the static case (as evidenced, from the onset, by the impossibility result of Cash and Tessaro [CT14]). Nevertheless, many, if not most, applications of SSE require dynamism. This state of affairs significantly hinders the applicability of local and page-efficient SSE.

While one work [MM17] targets local SSE in a dynamic setting, and has constant storage efficiency and locality, it has read efficiency $\mathcal{O}(L\log W)$, where L is the maximum list size. Further, [MM17] employs an ORAM-variant which incurs a heavy computational overhead, in addition to the large read efficiency. When reinterpreting [MM17] in the context of page-efficiency, its guarantees improve to $\mathcal{O}(\log W)$ page efficiency and constant storage efficiency, but the heavy computational cost of ORAM remains.

1.1 Our Contributions

In this article, we consider the problem of dynamic memory-efficient SSE, by which we mean that we target both dynamic *page-efficient* SSE, and dynamic *local* SSE.

The centerpiece of our approach is a novel connection between these two goals. We introduce a map, called the Generic Local Transform, which takes as input a page-efficient SSE scheme with certain special features, and outputs a SSE scheme with strong locality properties. Our strategy will be to first build page-efficient schemes, then apply the Generic Local Transform to obtain local schemes. This approach turns out to be quite effective, and we present several results.

(1) **Dynamic page-efficient SSE.** We start by building a dynamic page-efficient SSE scheme, LayeredSSE. LayeredSSE achieves storage efficiency $\mathcal{O}(1)$,

and page efficiency $\widetilde{\mathcal{O}}\left(\log\log\frac{N}{p}\right)$, where p is the page size. In line with prior work on memory-efficient SSE, the technical core of LayeredSSE is a new dynamic allocation scheme, L2C. L2C is a weighted variant of the so-called "2-choice" algorithm, notorious in the resource allocation literature. L2C is of independent interest: the two-choice allocation process is ubiquitous in various areas of computer science, such as load balancing, hashing, job allocation, or circuit routing (a survey of applications may be found in [RMS01]). Weighted variants have been considered in the past, but have so far required a *distributional* assumption [TW07, TW14] or presorting [ANSS16]. What we show is that by slightly tweaking the two-choice process, a dynamic and distribution-free result can be obtained (Theorem 1). Such a distribution-free result is necessary for cryptographic applications, where the adversary may influence the weights (as in our case). Other uses beyond cryptography are discussed in the full version.

(2) **Generic Local Transform.** We introduce the Generic Local Transform. On input any page-efficient scheme PE-SSE with certain special features, called *page-length-hiding* SSE, the Generic Local Transform outputs a local SSE scheme Local[PE-SSE]. Roughly speaking, if PE-SSE has client storage $\mathcal{O}(1)$, storage efficiency $\mathcal{O}(1)$, and page efficiency $\mathcal{O}(P)$, then Local[PE-SSE] has storage efficiency $\mathcal{O}(1)$, and read efficiency $\mathcal{O}(P)$. Regarding locality, the key feature is that if PE-SSE has locality $\mathcal{O}(L)$ *when querying lists of size at most one page*, then Local[PE-SSE] has locality $\mathcal{O}(L + \log\log N)$ *when querying lists of any size*. Thus, the Local construction may be viewed as bootstrapping a scheme with weak locality properties into a scheme with much stronger locality properties.

The Generic Local Transform also highlights an interesting connection between the goals of page efficiency and locality. Originally, locality and page efficiency were introduced as distinct performance criterions, targeting the two most widespread storage media, HDDs and SSDs respectively. It was already observed in [BBF+21] that a scheme with locality L and read efficiency R must have page efficiency at most $R + 2L$. In that sense, page efficiency is an "easier" goal. With the Generic Local Transform, surprisingly, we build a connection in the reverse direction: we use page-efficient schemes as building blocks to obtain local schemes. On a theoretical level, this shows a strong connection between the two goals. On a practical level, it provides a strategy to target both goals at once.

(3) **Dynamic local SSE.** By applying the Generic Local Transform to the LayeredSSE page-efficient scheme, we immediately obtain a dynamic SSE scheme Local[LayeredSSE], with storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\widetilde{\mathcal{O}}(\log\log N)$. The construction is conditional: it requires that the longest list is of size $\mathcal{O}\left(N^{1-1/\log\log N}\right)$. The asymptotic performance of Local[LayeredSSE] matches exactly the second *static* construction from [ANSS16], including the condition on maximum list size: dynamism comes at no extra cost. In particular, Local[LayeredSSE] matches the lower bound from [ASS21] for SSE schemes built using what [ASS21] refers to as "allocation schemes"—showing that the bound can be matched even in the dynamic setting.

(4) **Unconditional local SSE in the static setting.** The original 1-choice scheme from [ANSS16] achieves $\mathcal{O}(1)$ storage efficiency, $\mathcal{O}(1)$ locality, and $\widetilde{\mathcal{O}}(\log N)$ read efficiency, unconditionally. The read efficiency was improved to $\mathcal{O}\left(\log^{2/3+\varepsilon} N\right)$ in [DPP18], for any constant $\varepsilon > 0$. This was, until now, the only SSE construction to achieve sublogarithmic efficiency unconditionally. By applying the Generic Local Transform to a variant of Tethys [BBF+21], in combination with techniques inspired by [DPP18], we obtain an unconditional static SSE scheme with storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\mathcal{O}(\log^{\varepsilon} N)$, for any constant $\varepsilon > 0$. To our knowledge, this is the construction that comes closest to the impossibility result of Cash and Tessaro, stating that $\mathcal{O}(1)$ locality, storage efficiency, and read efficiency simultaneously is impossible.

Table 1 – Page-efficient SSE schemes. N denotes the total size of the database, W denotes the number of keywords, p is the number elements per page, $\varepsilon > 0$ is an arbitrarily small constant, and λ is the security parameter.

Schemes	Client st.	Page eff.	Storage eff.	Dynamism
$\Pi_{\text{pack}}, \Pi_{\text{2lev}}$ [CJJ+14]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$	Static
TCA [ANSS16]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log\log N)$	$\mathcal{O}(1)$	Static
Tethys [BBF+21]	$\mathcal{O}(p\log\lambda)$	3	$3+\varepsilon$	Static
IO-DSSE [MM17]	$\mathcal{O}(W)$	$\mathcal{O}(\log W)$	$\mathcal{O}(1)$	Dynamic
LayeredSSE	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}\left(\log\log \frac{N}{p}\right)$	$\mathcal{O}(1)$	Dynamic

Table 2 – SSE schemes with constant locality and storage efficiency. N denotes the total size of the database, and $\varepsilon > 0$ is an arbitrarily small constant.

Schemes	Locality	Read eff.	St. eff.	Max list size	Dynamism
TCA [ANSS16]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log\log N)$	$\mathcal{O}(1)$	$\mathcal{O}\left(N^{1-1/\log\log N}\right)$	Static
[ASS21]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log\log\log N)$	$\mathcal{O}(1)$	$\mathcal{O}\left(N^{1-1/\log\log\log N}\right)$	Static
OCA [ANSS16]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log N)$	$\mathcal{O}(1)$	Unconditional	Static
[DPP18]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}\left(\log^{2/3+\varepsilon} N\right)$	$\mathcal{O}(1)$	Unconditional	Static
Local[LayeredSSE]	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log\log N)$	$\mathcal{O}(1)$	$\mathcal{O}\left(N^{1-1/\log\log N}\right)$	Dynamic
UncondSSE	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log^{\varepsilon} N)$	$\mathcal{O}(1)$	Unconditional	Static

Remark on Forward Security. The SSE schemes built in this work have a standard "minimal" leakage profile during Search: namely, searches leak the search pattern, the access pattern and the length of the retrieved list of document identifiers. For our dynamic schemes, Update operations importantly leak no

information about unqueried keywords, but leak an identifier of the list being updated, as well as, in some cases, the length of the list. As a consequence, our dynamic schemes are not *forward-secure*. The underlying issue is that the goals of forward security and memory efficiency seem to be fundamentally at odds. Indeed, locality asks that identifiers associated to the same keywords must be stored close to each other; while forward-privacy requires that the location where a new identifier is inserted should be independent of the keyword it is associated with. That issue was already noted in [Bos16], who claims that "for dynamic schemes, locality and forward-privacy are two irreconcilable notions". We refer the reader to [Bos16] for more discussion of the problem and leave further analysis of this issue for future work.

Note that SSE has a very varied range of uses cases, for example private database services, online messaging and encrypted text search. In practice, its security requirements depend entirely on the use case. There are use cases where forward secrecy is crucial. The argument for forward security that is often given in the literature (e.g. [Bos16, BMO17, EKPE18, AKM19]) is to thwart file injection attacks in the style of [ZKP16]. Those attacks require injecting adversarially crafted entries into the target database. In an online messaging scenario, those attacks could be realistic, hence forward security is needed. In other cases, adversarial file injection is much less of a threat, and forward security can be reasonably dispensed with. For use cases where forward security is not required, we show that dynamism and memory efficiency are achievable at the same time.

Remark on the Focus on the Reverse Index. As most SSE literature, this work focuses on the (inverse) document index. The simplest usage scenario is to retrieve document indices from the index, then fetch those documents from a separate database. In reality, there are many other ways to use the index, for example by intersecting the document indices from several queries before fetching, fetching only some of the documents (see [MPC+18]), or building graph databases via several layers of inverse indices [CK10].

In most cases, the cost of fetching the actual documents is the same for the encrypted database as it is for the equivalent plaintext database: the efficiency overhead comes entirely from the inverse index. Schemes that hide access pattern or volume leakage are a possible exceptions but are out of the scope of this work.

2 Technical Overview

This work contains several results, tied together by the Generic Local Transform. As such, we believe it is beneficial to present them together within one paper. This requires introducing a number of different allocation mechanisms. We have endeavored to provide in this section a clear overview of those mechanisms. Formal specifications, theorems, and proofs will be presented in subsequent sections.

It is helpful to fist recall a few well-studied allocation mechanisms. In what follows, "with overwhelming probability" is synonymous with "except with negligible probability" (in the usual cryptographic sense), whereas "with high proba-

bility" simply means with probability close to 1 in some sense, but not necessarily overwhelming.

One-choice allocation. In one-choice allocation, n balls are thrown into n bins. Each ball is inserted into a bin chosen independently and uniformly at random (by hashing an identifier of the ball). A standard analysis using Chernoff bounds shows that, at the outcome of the insertion process, the most loaded bin contains $\mathcal{O}(\log n)$ balls with high probability [JK77]. (And at most $\mathcal{O}(f(n)\log n)$ balls with overwhelming probability, for any $f = \omega(1)$.)

Two-choice allocation. Once again, n balls are thrown into n bins. For each ball, two bins are chosen independently and uniformly at random (e.g. by hashing an identifier of the ball). The ball is inserted into whichever of the two bins contains the fewest balls at the time of insertion. A celebrated result by Azar et al. shows that, at the outcome of the insertion process, the most loaded bin contains $\mathcal{O}(\log \log n)$ balls with high probability [ABKU94]. (It was later shown that the result holds with overwhelming probability [RMS01].)

2.1 Layered 2-Choice Allocation

Our first goal is to build a dynamic page-efficient scheme. Let us summarize what this entails, starting with the static case. As explained in the introduction, to realize single-keyword SSE, we want to store lists of arbitrary sizes on an untrusted server. Hiding the contents of the lists can be achieved in a straightforward way using symmetric encryption. The main challenge is how to store the lists in the server memory, in such a way that accessing one list does not reveal information about the lengths of other lists.

In the case of page-efficient schemes, this challenge may be summarized as follows. We are given a set of lists, containing N items in total. We are also given a page size p, which represents the number of items that can fit within a physical memory page. The memory of the server is viewed as an array of pages. We want to store the lists in the server memory, with three goals in mind.

1. In order to store all lists, we use $S\lceil N/p \rceil$ pages of server memory in total, where S is called the *storage efficiency* of the allocation scheme. We want S to be as small as possible.
2. Any list of length ℓ can be retrieved by visiting at most $P\lceil \ell/p \rceil$ pages in server memory, where P is called the *page efficiency* of the allocation scheme. We want P to be as small as possible.
3. Finally, the pages visited by the server to retrieve a given list should not depend on the lengths of other lists.

The first two goals are precisely the aim of bin packing algorithms. The third goal is a security goal: it stipulates that the pattern of memory accesses performed by the server should not leak certain information. As such, the goal relates to oblivious or data-independent algorithms. In [BBF+21], a framework for realizing the three goals was formalized as *Data-Independent Packing* (DIP).

To ease presentation, we will focus on the case where all lists are of size at most one page. If a list is of length more than one page, the general idea is that it will be split into chunks of one page, plus one final chunk of size at most one page; each chunk will then be treated as a separate list by the allocation scheme. We assume from now on that lists are of length less than one page.

In a nutshell, the idea proposed by [BBF+21] to instantiate a DIP scheme is to use weighted variant of cuckoo hashing [PR04]. In more detail, for each list, two pages are chosen uniformly at random, by hashing an identifier of the list. Each element of the list will then be stored in one of the two designated pages, or a stash. The stash is stored on the client side. In order to choose how each list is split between its three possible destinations (the two chosen pages, or the stash), [BBF+21] uses a maximum flow algorithm. The details of this algorithm are not relevant for our purpose. The important point is that when retrieving a list, the server accesses two uniformly random pages. Clearly, this reveals no information to the server about the lengths of other lists. The resulting algorithm, called Tethys, achieves storage efficiency $\mathcal{O}(1)$, page efficiency $\mathcal{O}(1)$, with client storage $\omega(\log \lambda)$ pages (used to store the stash).

In this paper, we wish to build a dynamic SSE. For that purpose, the underlying allocation scheme needs to allow for a new *update* operation. An update operation allows the client to add a new item to a list, increasing its length by one. The security goal remains essentially the same as in the static case: the pages accessed by the algorithm in order to update a given list should not depend on the lengths of other lists.

Tethys is not a suitable basis for a dynamic scheme, because it does not allow for an efficient data-independent update procedure: when inserting an element into a cell, the update procedure requires running a max flow algorithm. This either requires accessing other cells, with an access pattern that is intrinsically data-dependent, or performing a prohibitively expensive data-oblivious max flow computation each update. Instead, a natural idea is to use a weighted variant of the two-choice allocation scheme. With two-choice allocation, the access pattern made during an update is simple: only the two destination buckets associated to the list being updated need to be read. The new item is then inserted into whichever of the two buckets currently contains less items.

Instantiating that approach would require a weighted *dynamic* variant of two-choice allocation, along the following lines: given a multiset of list sizes $\{\ell_i : 1 \leq i \leq k\}$ with $\ell_i \leq p$ and $\sum \ell_i = N$, at the outcome of a two-choice allocation process into $\mathcal{O}(N/p)$ buckets, the most loaded bucket contains $\mathcal{O}(p \log \log N)$ items with overwhelming probability, even if the weight of balls is updated during the process. However, a result of that form appears to be a long-standing open problem (some related partial results are discussed in [BFHM08]). The two-choice process with weighted items has been studied in the literature [TW07, TW14, ANSS16], but to our knowledge, all existing results assume that (1) either the weight of the balls are sampled identically and independently from a sufficiently smooth distribution or (2) the balls are sorted initially and then allocated in decreasing order. Even disregarding constraints

on the distribution, in our setting, we cannot even afford to assume that list lengths are drawn independently: in the SSE security model, lists are chosen and updated *arbitrarily* by the adversary. Also, presorting the lists according to their length is not possible in a dynamic setting, as the list lengths can be changed via updates.

For our purpose, we require a *distribution-free* statement: we only know a bound p on the size of each list, and a bound N on the total size of all lists. We want an $\mathcal{O}(p \log \log N)$ upper bound on the size of the most loaded bucket that holds for *any* set of list sizes satisfying those constraints, even if list sizes are updated during the process. A result of that form is known for one-choice allocation processes [BFHM08] (with a $\mathcal{O}(p \log N)$ upper bound), but the same article shows that the same techniques cannot extend to the two-choice process.

To solve that problem, we introduce a *layered* weighted 2-choice allocation algorithm, L2C. L2C has the same basic behavior as a (weighted) two-choice algorithm: for each ball, two bins are chosen uniformly at random as possible destinations. The only difference is how the bin where the ball is actually inserted is selected among the two destination bins. The most natural choice would be to store the ball in whichever bin currently has the least load, where the *load* of a bin is a the sum of the weights of the balls it currently contains. Instead, we use a slightly more complex decision process. In a nutshell, we partition the possible weights of balls into $\mathcal{O}(\log \log \lambda)$ subintervals, and the decision process is performed independently for balls in each subinterval. For the first subinterval (holding the smallest weights), we use a weighted one-choice process, while for the other subintervals, we use an unweighted two-choice process.

The point of this construction is that its analysis reduces to the analysis of the weighted one-choice process, and the unweighted two-choice process, for which powerful analytical techniques are known. We leverage those techniques to show that L2C achieves the desired distribution-free guarantees on the load of the most loaded bin. In practice, what this means is that we have an allocation algorithm that, for most intents and purposes, behaves like a weighted variant of two-choice allocation, and for which updates and distribution-free guarantees can be obtained relatively painlessly.

The LayeredSSE scheme is obtained by adding a layer of encryption and key management on top of L2C, using standard techniques from the SSE literature, although some care is required for updates. We refer the reader to Section 5 for more details.

2.2 Generic Local Transform

At Crypto 2018, Asharov et al. identified two main paradigms for building local SSE [ASS18]. The first is the *allocation* paradigm, which typically uses variants of multiple-choice allocation schemes, or cuckoo hashing. The second is the *pad-and-split* approach. The main difficulty of memory-efficient SSE is to pack together lists of different sizes. The idea of the pad-and-split approach is to store lists separately according to their size, which circumvents the issue. The simplest way to realize this is to pad all lists length to the next power of 2. This yields

$\log N$ possible values for list lengths. All lists of a given length can be stored together using, for instance, a standard hash table. Since we do not want to reveal the number of lists of each length, the hash table at each level needs to be dimensioned to be able to receive the entire database. As a result, a basic pad-and-split scheme has storage efficiency $\mathcal{O}(\log N)$, but easily achieves $\mathcal{O}(1)$ locality and read efficiency.

For the Generic Local Transform, we introduce the notion of *Overflowing SSE* (OSSE). An OSSE behaves like an SSE scheme in all aspects, except that, during its setup and during updates, it may refuse to store some list elements. Such elements are called *overflowing*. An OSSE is intended to be used as a subcomponent within an overarching SSE construction. The OSSE scheme is used to store part of the database, while overflowing elements are stored using a separate mechanism. The notion of OSSE was not formalized before, but in hindsight, the use of OSSE may be viewed as implicit in several existing constructions [DPP18, ASS18, BBF+21]. We choose to introduce it explicitly here for ease of exposition.

We are now in a position to explain the Generic Local Transform. The chief limitation of the pad-and-split approach is that it creates a $\log N$ overhead in storage. The high-level idea of the Generic Local Transform, then, is to use an OSSE to store all but a fraction $1/\log N$ of the database. Then a pad-and-split variant is used to store the $N/\log N$ overflowing elements. The intent is to benefit from the high efficiency of the pad-and-split approach, without having to pay for the $\log N$ storage overhead.

There is, however, a subtle but important issue with that approach. A given list may be either entirely stored within the OSSE scheme, or only partially stored, or not stored at all. In the OSSE scheme that we will later use (as well as OSSEs that were implicit in prior work), those three situations should be indistinguishable to the server, or else security breaks down. To address that issue, we proceed as follows.

Let us assume all lists have been padded to the next power of 2. For the pad-and-split part of the construction, we create $\log N$ SSE instances, one for each possible list size. We call each of these instances a *layer*. The overflowing elements of a list of size ℓ will be stored in the layer that handles lists of size ℓ, regardless of how many elements did overflow from the OSSE for that list.

The OSSE guarantees that the total number of overflowing items is at most $n = \mathcal{O}(N/\log N)$. Thus, if we focus on the layer that handles lists of size ℓ, the layer will receive at most n elements. These elements will be split into lists of size at most ℓ (corresponding to the set of overflowing elements, for each list of size ℓ in the original database). To achieve storage efficiency $\mathcal{O}(S)$ overall, we want the layer to store those lists using $\mathcal{O}(Sn)$ storage. To achieve read efficiency R, the layer should also be able to retrieve a given list by visiting at most $R\ell$ memory locations. This is where everything comes together: an SSE scheme satisfying those conditions is precisely a page-efficient SSE scheme with page size ℓ, storage efficiency S, and page efficiency R.

The page-efficient scheme used for each layer is also required satisfy a few extra properties: first, when searching for a list of size at most one page, the length of the list should not be leaked. We call this property *page-length-hiding*. (We avoid the term *length-hiding* to avoid confusion with volume-hiding SSE, which fully hides lengths.) All existing page-efficient constructions have that property. Second, we require the page-efficient scheme to have $\mathcal{O}(1)$ client storage. All constructions in this article satisfy that property, but the construction from [BBF+21] does not. Finally, we require the scheme to have locality $\mathcal{O}(1)$ when fetching a single page. All existing page-efficient constructions have this property. (The last two properties could be relaxed, at the cost of more complex formulas and statements.) We call an SSE scheme satisfying those three properties *suitable*.

Putting everything together, the Generic Local Transform takes as input a suitable *page-efficient* scheme, with storage efficiency S and page efficiency P. It outputs a *local* scheme with storage efficiency $S + S'$, read efficiency $P + R'$, and locality L', where S', R', and L' are the storage efficiency, read efficiency, and locality of the underlying OSSE. It remains to explain how to build a local OSSE scheme with $\mathcal{O}(N/\log N)$ overflowing items, discussed next.

2.3 ClipOSSE: an OSSE scheme with $\mathcal{O}(N/\log N)$ Overflowing Items

At STOC 2016, Asharov et al. introduced so-called "2-dimensional" variants of one-choice and two-choice allocation, for the purpose of building local SSE. The one-choice variant works as follows. Consider an SSE database with N elements. Allocate $m = \widetilde{\mathcal{O}}(N/\log N)$ buckets, initially empty. For each list of length ℓ in the database, choose one bucket uniformly at random. The first element of the list is inserted into that bucket. The second element of the list is inserted into the next bucket (assuming a fixed order of buckets, which wraps around when reaching the last bucket), the third one into the bucket after that, and so on, until all list elements have been inserted. Thus, assuming $\ell \leq m$, all list elements have been placed into ℓ consecutive buckets, one element in each. An analysis very similar to the usual analysis of the one-choice process shows that with overwhelming probability, the most loaded bucket receives at most $\tau = \widetilde{\mathcal{O}}(\log N)$ elements. To build a static SSE scheme from this allocation scheme, each bucket is padded to the maximal size τ and encrypted. Search queries proceed in the natural way.

Such a scheme yields storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$ (since retrieving a list amounts to reading consecutive buckets), and read efficiency $\widetilde{\mathcal{O}}(\log N)$ (since retrieving a list of length ℓ requires reading ℓ buckets, each of size $\tau = \widetilde{\mathcal{O}}(\log N)$). To build ClipOSSE, we start from the same premise, but "clip" buckets at the threshold $\tau = \widetilde{\mathcal{O}}(\log \log N)$. That is, each bucket can only receive up to τ elements. Elements that cannot fit are overflowing.

In the standard one-choice process, where n balls are thrown i.i.d. into n bins, it is not difficult to show that clipping bins at height $\tau = \mathcal{O}(\log \log n)$ results

in at most $\mathcal{O}(n/\log n)$ overflowing elements with overwhelming probability. In fact, by adjusting the multiplicative constant in the choice of τ, the number of overflowing elements can be made $\mathcal{O}\left(n/\log^d n\right)$ for any given constant d. We show that a result of that form still holds for (a close variant of) the 2-dimensional one-choice process outlined earlier. The result is conditional: it requires that the maximum list size is $\mathcal{O}(N/\mathsf{polylog}\, N)$. (A condition of that form is necessary, insofar as the result fails when the maximum list size gets close to $N/\log N$.) The proof of the corresponding theorem is the most technically challenging part of this work, and relies on the combination of a convexity argument with a stochastic dominance argument. An overview of the proof is given in section 6.5, so we omit more discussion here.

In the end, ClipOSSE achieves storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\mathcal{O}(\log\log N)$, with $\mathcal{O}\left(N/\log^d N\right)$ overflowing elements (for any fixed constant d of our choice), under the condition that the maximum list size is $\mathcal{O}(N/\mathsf{polylog}\, N)$. All applications of the Generic Local Transform in this article use ClipOSSE as the underlying OSSE. (That is why we write Local[PE-SSE] for the Generic Local Transform applied to the page-efficient scheme PE-SSE, and do not put the underlying OSSE as an explicit parameter.)

2.4 Dynamic Local SSE with $\widetilde{\mathcal{O}}(\log\log N)$ Overhead

By using the Generic Local Transform with ClipOSSE as the underlying OSSE, and LayeredSSE as the page-efficient scheme, we obtain Local[LayeredSSE]. The Local[LayeredSSE] scheme has storage efficiency $\mathcal{O}(1)$, locality $\mathcal{O}(1)$, and read efficiency $\widetilde{\mathcal{O}}(\log\log N)$. This result follows from the main theorem regarding the Generic Local Transform, and does not require any new analysis.

Local[LayeredSSE] is a conditional scheme: it requires that the longest list is of length $\mathcal{O}\left(N^{1-1/\log\log\lambda}\right)$. The reason is subtle. ClipOSSE by itself has a condition that the longest list is $\mathcal{O}(N/\mathsf{polylog}\, N)$, which is less demanding. The reason for the condition comes down to the fact that LayeredSSE only achieves a negligible probability of failure as long as the number of pages in the scheme is at least $\Omega(\lambda^{1/\log\log\lambda})$. More generally, the same holds for the number of bins in two-choice allocation processes in general, even the standard, unweighted process. The condition is optimal: [ASS21] shows that any sublogarithmic "allocation-based" scheme must be conditional, and gives a bound on the condition. Local[PE-SSE] matches that bound.

2.5 Unconditional Static Local SSE with $\mathcal{O}(\log^\varepsilon N)$ Overhead

The (static) Tethys scheme from [BBF+21] achieves storage efficiency $\mathcal{O}(1)$ and page efficiency $\mathcal{O}(1)$ simultaneously. It is also page-length-hiding. Since we have the Generic Local Transform at our disposal, it is tempting to apply it to Tethys. There is, however, one obstacle: Tethys uses $\omega(p\log\lambda)$ client memory, in order to store a stash on the client side. For the Generic Local Transform, we need

$\mathcal{O}(1)$ client memory. To reduce the client memory of Tethys, a simple idea is to store the stash on the server side. Naively, reading the stash for every search would increase the page efficiency to $\omega(\log \lambda)$. To avoid this, we store the stash within an ORAM.

For that purpose, we need an ORAM with a failure probability of zero: indeed, since we may store as few as $\log \lambda$ elements in the ORAM, a correctness guarantee of the form $\mathsf{negl}(n)$, where $n = \log \lambda$ is the number items in the ORAM, fails to be sufficient (it is not $\mathsf{negl}(\lambda)$). We also need the ORAM to have $\mathcal{O}(1)$ locality. An ORAM with these characteristics was devised in [DPP18], motivated by the same problem. The ORAM from [DPP18] achieves read efficiency $\mathcal{O}\left(n^{1/3+\varepsilon}\right)$, for any arbitrary constant $\varepsilon > 0$. It was already conjectured in [DPP18] that it could be improved to $\mathcal{O}(n^{\varepsilon})$. We build that variant explicitly, and name it $\mathsf{LocORAM}$. Roughly speaking, $\mathsf{LocORAM}$ is a variant of the Goldreich-Ostrovsky hierarchical ORAM, with a constant number of levels.

By putting the stash of Tethys within $\mathsf{LocORAM}$ on the server side, we naturally obtain a page-efficient SSE scheme $\mathsf{OramTethys}$, with $\mathcal{O}(\log^{\varepsilon} \lambda)$ read efficiency, suitable for use within the Generic Local Transform. This yields a static local SSE for lists of size at most $N/\mathsf{polylog}\, N$. To handle larger lists, borrowing some ideas from [DPP18], we group lists by size, and use again $\mathsf{OramTethys}$ to store them. In the end, we obtain an unconditional SSE with $\mathcal{O}(1)$ store efficiency, $\mathcal{O}(1)$ locality, and $\mathcal{O}(\log^{\varepsilon} \lambda)$ read efficiency.

Comparing with the $\mathcal{O}\left(\log^{2/3+\varepsilon} \lambda\right)$ construction from [DPP18], we note that the bottleneck of their construction comes from the allocation schemes the authors use for what they call "small" and "medium" lists. This is precisely the range where we use $\mathsf{Local}[\mathsf{OramTethys}]$. Our construction essentially removes that bottleneck, so that the $\mathcal{O}(\log^{\varepsilon} \lambda)$ read efficiency bottlneck now comes entirely from the ORAM component. A detailed description of the scheme is given in the full version.

3 Preliminaries

Let $\lambda \in \mathbb{N}$ be the security parameter. For a probability distribution X, we denote by $x \leftarrow X$ the process of sampling a value x from the distribution. Further, we say that x is We denote by $[a, b]_{\mathbb{R}}$ the interval $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ and extend this naturally to intervals of the form $[a, b)_{\mathbb{R}}, (a, b]_{\mathbb{R}}, (a, b)_{\mathbb{R}}$.

3.1 Symmetric Searchable Encryption

A database $\mathsf{DB} = \{w_i, (\mathsf{id}_1, \ldots, \mathsf{id}_{\ell_i})\}_{i=1}^{W}$ is a set of keyword-identifier pairs with W keywords. We assume that each keyword w_i is represented by a machine word of $\mathcal{O}(\lambda)$ bits. We write $\mathsf{DB}(w_i) = (\mathsf{id}_1, \ldots, \mathsf{id}_{\ell_i})$ for the list of identifiers matching w_i. Throughout the article, we set $N = \sum_{i=1}^{W} \ell_i$ and define p as the page size (which we treat as a variable, independent of the size of the database N).

A dynamic searchable symmetric encryption scheme Σ is a 4-tuple of PPT algorithms $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ such that

- $\Sigma.\mathsf{KeyGen}(1^\lambda)$: Takes as input the security parameter λ and outputs client secret key K.
- $\Sigma.\mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$: Takes as input the client secret key K, an upper bound on the database size N and a database DB. Outputs encrypted database EDB and client state st.
- $\Sigma.\mathsf{Search}(\mathsf{K}, w, \mathsf{st}; \mathsf{EDB})$: The client receives as input the secret key K, keyword w and state st. The server receives as input the encrypted database EDB. Outputs some data d and updated state st' for the client. Outputs updated encrypted database EDB' for the server.
- $\Sigma.\mathsf{Update}(\mathsf{K}, (w, L), \mathsf{op}, \mathsf{st}; \mathsf{EDB})$: The client receives as input the secret key K, a pair (w, L) of keyword w and list L of identifiers, an operation $\mathsf{op} \in \{\mathsf{del}, \mathsf{add}\}$ and state st. The server receives as input the encrypted database EDB. Outputs updated state st' for the client. Outputs updated encrypted database EDB' for the server.

In the following, we omit the state st and assume that it is implicitly stored and updated by the client. We say that Σ is *static*, if it does not provide an Update algorithm. Further, we assume that the keyword w is preprocessed via a $\mathsf{PRF}$by the client, whenever the client sends w to the server in either Search or Update. This ensures that the server never has access to w in plaintext and unqueried keywords are distributed uniformly random in the view of the server.

Intuitively, the client uses Setup to encrypt and outsource a database DB to the server. Then, the client can search keywords w using Search and receives the list of matching identifiers $\mathsf{DB}(w)$ from the server. The list $\mathsf{DB}(w)$ can be updated via Update, provided that the size of the database stays below N. Note that we allow the client to add (or delete) multiple identifiers at once for a single keyword (which is required for the Generic Local Transform section 6).

Security. We now define correctness and semantic security of SSE. Intuitively, correctness guarantees that a search always retrieves all matching identifiers and semantic security guarantees that the server only learns limited information (quantified by a leakage function) from the client.

Definition 1 (Correctness). *A* SSE *scheme* Σ *is correct if for all databases* DB *and* $N \in \mathbb{N}$*, keys* $\mathsf{K} \leftarrow \Sigma.\mathsf{KeyGen}(1^\lambda)$*,* $\mathsf{EDB} \leftarrow \Sigma.\mathsf{Setup}(\mathsf{K}, \mathsf{DB})$ *and sequences of search, add or delete queries* S*, the search protocol returns the correct result for all queries of the sequence, if the size of the database remains at most* N*.*

We use the standard semantic security notion for SSE (see [CGKO06]). Security is parameterized by a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$, composed of the setup leakage $\mathcal{L}_{\mathsf{Stp}}$, the search leakage $\mathcal{L}_{\mathsf{Srch}}$, and the update leakage $\mathcal{L}_{\mathsf{Updt}}$. We define two games, SSEReal and SSEIdeal. First, the adversary chooses a database DB. In SSEReal, the encrypted database EDB is generated by $\mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$, whereas in SSEIdeal the encrypted database is simulated by a (stateful) simulator Sim on input $\mathcal{L}_{\mathsf{Stp}}(\mathsf{DB}, N)$. After receiving EDB, the adversary issues search and update queries. All queries are answered honestly in

SSEReal. In SSEIdeal, the search queries on keyword w are simulated by Sim on input $\mathcal{L}_{\mathsf{Srch}}(w)$, and update queries for operation op, keyword w and identifier list L are simulated by Sim on input $\mathcal{L}_{\mathsf{Updt}}(\mathsf{op}, w, L)$. Finally, the adversary outputs a bit b.

We write $\text{SSEReal}^{\mathsf{adp}}$ and $\text{SSEIdeal}^{\mathsf{adp}}$ if the queries of the adversary were chosen adaptively, *i.e.* dependant on previous queries. Similarly, we write $\text{SSEReal}^{\mathsf{sel}}$ and $\text{SSEIdeal}^{\mathsf{sel}}$ if the queries are chosen selectively by the adversary, *i.e.* sent initially in conjunction with the database before receiving EDB.

Definition 2 (Semantic Security). *Let Σ be a SSE scheme and $\mathcal{L} = (\mathcal{L}_{\mathsf{Stp}}, \mathcal{L}_{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Updt}})$ a leakage function. Scheme Σ is $\mathcal{L}$-adaptively secure if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator Sim such that*

$$|\Pr[\text{SSEReal}^{\mathsf{adp}}_{\Sigma,\mathcal{A}}(\lambda) = 1] - \Pr[\text{SSEIdeal}^{\mathsf{adp}}_{\Sigma,\mathsf{Sim},\mathcal{L},\mathcal{A}}(\lambda) = 1]| = \mathsf{negl}(\lambda).$$

Similarly, scheme Σ is $\mathcal{L}$-selectively secure if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator Sim such that

$$|\Pr[\text{SSEReal}^{\mathsf{sel}}_{\Sigma,\mathcal{A}}(\lambda) = 1] - \Pr[\text{SSEIdeal}^{\mathsf{sel}}_{\Sigma,\mathsf{Sim},\mathcal{L},\mathcal{A}}(\lambda) = 1]| = \mathsf{negl}(\lambda).$$

Intuitively, semantic security guarantees that the interaction between client and server reveals no information to the server, except the leakage of the given query. The schemes from this article have common leakage patterns. We use the standard notions of query pattern qp and history Hist from [Bos16] to formalize this leakage: (1) The query pattern $\mathsf{qp}(w)$ for a keyword w are the indices of previous search or update queries for keyword w. (3) The history $\mathsf{Hist}(w)$ is comprised of the list of identifiers matching keyword w that were inserted during setup and the history of updates on keyword w, that is each deleted and inserted identifier. We can retrieve the number ℓ_i of inserted identifiers and the number d_i of deleted identifiers from $\mathsf{Hist}(w)$ for each keyword.

We define two leakage patterns we use throughout the article. (1) We define *page-length hiding* leakage $\mathcal{L}_{\mathsf{len\text{-}hid}}$. We set $\mathcal{L}_{\mathsf{len\text{-}hid}} = (\mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Srch}}, \mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Updt}})$, where the setup leakage is $\mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Stp}}(\mathsf{DB}, N) = N$ is the maximal size N of the database, the search leakage $\mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Srch}}(w) = (\mathsf{qp}, \lceil \ell_i/p \rceil, \lceil d_i/p \rceil)$ is the query pattern and the number of pages required to store the inserted and deleted items, and the update leakage $\mathcal{L}^{\mathsf{len\text{-}hid}}_{\mathsf{Updt}}(\mathsf{op}, w, L) = (\mathsf{op}, \mathsf{qp}, \lceil (\ell_i + |L|)/p \rceil, \lceil (d_i + |L|)/p \rceil, \lceil \ell_i/p \rceil, \lceil d_i/p \rceil)$ is the operation, the query pattern and the number of pages required to store the inserted and deleted items (before and after the update)[1]. (2) Similarly, we define *length reveiling* leakage $\mathcal{L}_{\mathsf{len\text{-}rev}}$. We set $\mathcal{L}_{\mathsf{len\text{-}rev}} = (\mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Stp}}, \mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Srch}}, \mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Updt}})$ with $\mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Stp}}(\mathsf{DB}, N) = N$, $\mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Srch}}(w) = (\mathsf{qp}, |L'|, \ell_i, d_i)$ and lastly $\mathcal{L}^{\mathsf{len\text{-}rev}}_{\mathsf{Updt}}(\mathsf{op}, w, L') = (\mathsf{op}, \mathsf{qp}, |L'|, \ell_i, d_i)$.

We will use $\mathcal{L}_{\mathsf{len\text{-}hid}}$ and $\mathcal{L}_{\mathsf{len\text{-}rev}}$ for both dynamic and static schemes. When we say that a static scheme is $\mathcal{L}$-semantically secure, for $\mathcal{L} \in \{\mathcal{L}_{\mathsf{len\text{-}hid}}, \mathcal{L}_{\mathsf{len\text{-}rev}}\}$, we

[1] Note that we allow for inserting more than one identifier per keyword in a single update operation in this work. Thus, the server will also learn (limited) information about the number $|L|$ of added or deleted identifiers.

simply ignore the update leakage. Note that both leakage patterns, $\mathcal{L}_{\mathsf{len\text{-}hid}}$ and $\mathcal{L}_{\mathsf{len\text{-}rev}}$, have standard setup and search leakage, common in most SSE schemes. The update leakage of $\mathcal{L}_{\mathsf{len\text{-}hid}}$ and $\mathcal{L}_{\mathsf{len\text{-}rev}}$ is similar to their search leakage, and reveals nothing about unqueried keywords. While the update leakage is not forward secure, similar leakage patterns are commonly considered in literature, for example [CJJ+14]. We hope our techniques pave the way for future work on dynamic schemes with forward security and memory efficiency.

Efficiency Measures. We recall the notions of locality, storage efficiency and read efficiency [CT14], and page efficiency [BBF+21] (and extend them to the dynamic SSE setting in a natural manner). In the following definitions, we set $\mathsf{K} \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $\mathsf{EDB} \leftarrow \mathsf{Setup}(\mathsf{K}, N, \mathsf{DB})$ given database DB and upper bound N on the number of document identifiers. Also, $S = (\mathsf{op}_i, \mathsf{in}_i)_{i=1}^s$ is a sequence of search and update queries, where $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}, \bot\}$ is a operation and $\mathsf{in}_i = (\mathsf{op}_i, w_i, L_i, \mathsf{st}_i, \mathsf{EDB}_i)$ its input. Here, w_i is a keyword and L_i is a (added or deleted) list of identifiers, and after executing all previous operations op_j for $j \leq i$, st_i is the client state and EDB_i the encrypted database. We denote by DB_i the database after i operations. We assume that the total number of identifiers never exceeds N. (If $\mathsf{op}_i = \bot$, the query is a search query and L_i is empty.)

Definition 3 (Read Pattern). *Regard server-side storage as an array of memory locations, containing the encrypted database* EDB. *When processing search query* $\mathsf{Search}(\mathsf{K}, w_i, \mathsf{st}_i; \mathsf{EDB}_i)$ *or update query* $\mathsf{Update}(\mathsf{K}, (w_i, L_i), \mathsf{op}_i, \mathsf{st}_i; \mathsf{EDB}_i)$, *the server accesses memory locations* $m_1, \ldots, m_h$. *We call these locations the* read pattern *and denote it with* $\mathsf{RdPat}(\mathsf{op}_i, \mathsf{in}_i)$.

Definition 4 (Locality). *A SSE scheme has locality* L *if for any* λ, DB, N, *sequence* S, *and any* i, $\mathsf{RdPat}(\mathsf{op}_i, \mathsf{in}_i)$ *consists of at most* L *disjoint intervals.*

Definition 5 (Read Efficiency). *A SSE scheme has read efficiency* R *if for any* λ, DB, N, *sequence* S, *and any* i, $|\mathsf{RdPat}(\mathsf{op}_i, \mathsf{in}_i)| \leq R \cdot P$, *where* P *is the number of memory locations needed to store all (added and deleted) document indices matching keyword* w_i *in plaintext (by concatenating indices).*

Definition 6 (Storage Efficiency). *A SSE scheme has storage efficiency* E *if for any* λ, DB, N, *sequence* S, *and any* i, $|\mathsf{EDB}_i| \leq E \cdot |DB_i|$.

Definition 7 (Page Pattern). *Regard server-side storage as an array of pages, containing the encrypted database* EDB. *When processing search query* $\mathsf{Search}(\mathsf{K}, w_i, \mathsf{st}_i; \mathsf{EDB}_i)$ *or update query* $\mathsf{Update}(\mathsf{K}, (w_i, L_i), \mathsf{op}_i, \mathsf{st}_i; \mathsf{EDB}_i)$, *the read pattern* $\mathsf{RdPat}(\mathsf{op}_i, \mathsf{in}_i)$ *induces a number of page accesses* $p_1, \ldots, p_{h'}$. *We call these pages the* page pattern, *denoted by* $\mathsf{PgPat}(\mathsf{op}_i, \mathsf{in}_i)$.

Definition 8 (Page Cost). *A SSE scheme has page cost* $aX + b$, *where* a, b *are real numbers, and* X *is a fixed symbol, if for any* λ, DB, N, *sequence* S, *and any* i, $|\mathsf{PgPat}(\mathsf{op}_i, \mathsf{in}_i)| \leq aX + b$, *where* X *is the number of pages needed to store document indices matching keyword* w_i *in plaintext.*

Definition 9 (Page Efficiency). *A SSE scheme has page efficiency P if for any λ, DB, N, sequence S, and any i, $|\mathsf{PgPat}(\mathsf{op}_i, \mathsf{in}_i)| \leq P \cdot X$, where X is the number of pages needed to store document indices matching keyword w_i in plaintext.*

4 Layered Two-Choice Allocation

In this section, we describe layered two-choice allocation (L2C), a variant of two-choice allocation that allows to allocate n weighted balls (b_i, w_i) into m bins, where b_i is a unique identifier and $w_i \in [0,1]_{\mathbb{R}}$ is the weight of the ball. (We often write ball b_i for short.) First, let $1 \leq \delta(\lambda) \leq \log(\lambda)$ be a function. We denote by $w = \sum_{i=1}^{n} w_i$ the sum of all weights and set $m = w/(\delta(\lambda) \log\log w)$. We will later choose $\delta(\lambda) = o(\log\log\lambda)$ such that allocation has negligible failure probability. In the overview, we set $\delta(\lambda) = 1$ and assume that $m = \Omega(\lambda)$ for simplicity (which suffices for negligible failure probability).

Overview of L2C. L2C is based on both *weighted* one-choice allocation (1C) and *unweighted* two-choice allocation (2C). On a high level, we split the set of possible weights $[0,1]_{\mathbb{R}}$ into $\log\log m$ subintervals

$$[0, 1/\log m]_{\mathbb{R}}, (1/\log m, 2/\log m]_{\mathbb{R}}, \ldots, (2^{\log\log m - 1}/\log m, 1]_{\mathbb{R}}.$$

In words, the first interval is of size $1/\log m$ and the boundaries between intervals grow by a factor 2 every time. We will allocate balls with weights in a given subinterval independently from the others.

Balls in the first subinterval have weights $w_i \leq \log m$ and are thus small enough to apply weighted 1C. Intuitively, this suffices because one-choice (provably) performs worst for uniform weights of maximal size $1/\log m$. In that case, there are at most $n' = w \log m$ balls and we expect a bin to contain $n'/m = \log m \cdot \log\log w$ balls of uniform weight, since $m = w/(\log\log w)$. As each ball has weight $1/\log m$, the expected load per bin is $\log\log w$. This translates to a $\mathcal{O}(\log\log w)$ bound with overwhelming probability after applying a Chernoff's bound.

For the other intervals, applying unweighted and independent 2C per interval suffices, as the weights of balls differ at most by a factor 2 and there are only $\log\log m$ intervals. More concretely, let n_i be the number of balls in the i-th subinterval $A_i = (2^{i-1}/\log m, 2^i/\log m]_{\mathbb{R}}$ for $i \in \{1, \ldots, \log\log m\}$. Balls with weights in subinterval A_i fill the bins with at most $\mathcal{O}(n_i/m + \log\log m)$ balls, independent of other subintervals. Note that we are working with small weights, and thus potentially have $\omega(m)$ balls. Thus, we need to extend existing 2C results to negligible failure probability in m for the heavily-loaded case. As there are only $\log\log m$ subintervals, and balls in interval A_i have weight at most $2^i/\log m$, we can just sum the load of each subinterval and receive a bound

$$\sum_{i=1}^{\log\log m} \frac{2^i}{\log m} \mathcal{O}(n_i/m + \log\log m) = \mathcal{O}(w/m + \log\log m).$$

In total, we have $\mathcal{O}(w/m + \log\log m) = \mathcal{O}(\log\log w)$ bounds for the first and the remaining intervals. Together, this shows that all bins have load at most $\mathcal{O}(\log\log w)$ after allocating all n items. This matches the bound of standard 2C with unweighted balls if $m = \Omega(\lambda)$. For our SSE application, we want to allow for negligible failure probability with the least number of bins possible. We can set $\delta(\lambda) = \log\log\log(\lambda)$ and obtain a bin size of $\widetilde{\mathcal{O}}(\log\log w)$ with overwhelming probability, if $m = \frac{w}{\delta(\lambda)\log\log w}$. The analysis is identical in this case.

Handling Updates. The described variant of L2C is static. That is, we have not shown a bound on the load of the most loaded bin if we add balls or update the weight of balls. Fortunately, inserts of new balls are trivially covered by the analysis sketched above, if m was chosen large enough initially in order to compensate for the added weight. Thus, we assume there is some upper bound w_{max} on the total weights of added balls which is used to initially set up the bins. We can also update weights if we proceed with care.

For this, let b_i be some ball with weight w_{old}. We want to update its weight to $w_{\mathsf{new}} > w_{\mathsf{old}}$. If w_{old} and w_{new} reside in the subinterval, we can directly update the weight of b_i, as L2C ignores the concrete weight of balls inside a given subinterval for the allocation. Indeed, in the first interval, the bin in which b_i is inserted is determined by a single random choice, and for the remaining subintervals, the 2C process only considers the number of balls inside the same subinterval, ignoring concrete weights.

When w_{new} is larger than the bounds of the current subinterval, we need to make sure that the ball is inserted into the correct bin of its two choices. For this, the ball b_i is inserted into the bin with the lowest number of balls with weights inside the new subinterval. Even though the bin of b_i might change in this process, we still need to consider b_i as a ball of weight w_{old} in the old bin for subsequent ball insertions in the old subinterval. Thus, we mark the ball as *residual* ball but do not remove it from its old bin. That is, we consider it as ball of weight w_{old} for the 2C process but assume it is not identified by b_i anymore. As there are only $\log\log m$ different subintervals, storing the residual balls has a constant overhead. The full algorithm L2C is given in Algorithm 1. We parameterize it by a hash function H mapping uniformly into $\{1, \ldots, m\}^2$. The random bin choices of a ball b_i are given by $\alpha_1, \alpha_2 \leftarrow \mathsf{H}(b_i)$.

Load Analysis of L2C. Let either $\delta(\lambda) = 1$ or $\delta(\lambda) = \log\log\log\lambda$ and m sufficiently large such that $m^{-\Omega(\delta(\lambda)\log\log w)} = \mathsf{negl}(\lambda)$. (Note that this is the probability that allocation of 1C and 2C fails.)

We need to show that after setup and during a (selective) sequence of operations, the most loaded bin has a load of at most $\mathcal{O}(\delta(\lambda)\log\log w_{\mathsf{max}})$, where w_{max} is an upper bound on the total weight of the inserted balls. We sketch the proof here and refer to the full version for further details. First, we modify the sequence S such that we can reduce the analysis to only (sufficiently independent) L2C.InsertBall operations, while only increasing the final bin load by a constant factor. This is constant factor of the load is due to the additional weight of residual balls. Then, we analyze the load of the most loaded bin for the

each subinterval independently. This boils down to an analysis of a 1C process in the first subinterval and a 2C process in the remaining subintervals as in the overview of L2C (see Section 4). Summing up the independent bounds yields the desired result.

Theorem 1. *Let either* $\delta(\lambda) = 1$ *or* $\delta(\lambda) = \log\log\log\lambda$. *Let* $w_{\max} = \mathsf{poly}(\lambda)$ *and* $m = w_{\max}/(\delta(\lambda)\log\log w_{\max})$. *We require that* $m = \Omega(\lambda^{\frac{1}{\log\log\lambda}})$ *if* $\delta(\lambda) = \log\log\log\lambda$ *or* $m = \Omega(\lambda)$ *otherwise. Let* $\{(b_i, w_i)_{i=1}^n\}$ *be balls with (pair-wise unique) identifier* b_i *and weight* $w_i \in [0,1]$. *Further, let* $S = (\mathsf{op}_i, \mathsf{in}_i)_{i=n+1}^{s+n}$ *be a sequence of* s *insert or update operations* $\mathsf{op}_i \in \{\mathit{L2C}.\mathsf{InsertBall}, \mathit{L2C}.\mathsf{UpdateBall}\}$ *with input* $\mathsf{in}_i = (b_i, w_i, B_{\alpha_{i,1}}, B_{\alpha_{i,2}})$ *for inserts and* $\mathsf{in}_i = (b_i, o_i, w_i, B_{\alpha_{i,1}}, B_{\alpha_{i,2}})$ *for updates. Here,* b_i *denotes the identifier of a ball with weight* w_i *and old weight* $o_i \leq w_i$ *before the execution of* op_i. *Also, the bins are chosen via* $\alpha_{i,1}, \alpha_{i,2} \leftarrow \mathsf{H}(b_i)$.

Execute $(B_i)_{i=1}^m \leftarrow \mathit{L2C}.\mathsf{Setup}(\{(b_i, w_i)_{i=1}^n\})$ *and the operations* $\mathsf{op}_i(\mathsf{in}_i)$ *for all* $i \in [n+1, n+s]$. *We require that* $\sum_{i=1}^{n+s} w_i - o_i \leq w_{\max}$, i.e. *the total weight after all operations is at most* $w_{\max}$.

Then it holds that throughout the process, the most loaded bin of $B_1, \ldots, B_m$ *has at most load* $\mathcal{O}(\delta(\lambda)\log\log w_{\max})$ *except with negligible probability, if* H *is modeled as a random oracle.*

5 Dynamic Page Efficient SSE

We introduce the SSE scheme LayeredSSE based on L2C. Essentially, we interpret lists L_i of identifiers matching keyword w_i as balls of a certain weight. Then, we use L2C to manage the balls in m encrypted bins, where each bin corresponds to a memory page, yielding page efficiency $\widetilde{\mathcal{O}}(\log\log N/p)$ and constant storage efficiency. Let N be the maximal size of the database, $p \leq N^{1-1/\log\log\lambda}$ be the page size[2] and H be a hash function mapping into $\{1, \ldots, m\}^2$ for $m = \lceil w_{\max}/(\log\log\log\lambda \cdot \log\log w_{\max})\rceil$ and $w_{\max} = N/p$. Due to space limitations, we assume that each keyword has at most p associated keywords, and outline the scheme and its security analysis. We refer to the full version for details (without restrictions on the database[3]).

For convenience, we adapt the notation of L2C to lists of identifiers. A ball (w, L) of weight $|L|/p \in [0,1]_{\mathbb{R}}$ is a list of (at most p) identifiers matching keyword w. The 2 bin choices α_1, α_2 for ball (w, L) are given via $(\alpha_1, \alpha_2) \leftarrow \mathsf{H}(w)$. Now, L2C.Setup takes input balls $\{(w_i, L_i)\}_{i=1}^W$ and maximal weight $w_{\max}$, and allocates them as before into m bins. L2C.InsertBall receives ball (w, L) and two bins $(B_{\alpha_1}, B_{\alpha_2})$, and inserts (w, L) into either bin B_{α_1} or bin B_{α_2} as before.

[2] This condition is needed for the requirement $m \geq \lambda^{1/\log\log\lambda}$ of L2C which guarantees negligible failure probability (see Theorem 1). In practice, we have $p \ll N$.

[3] For arbitrary lists sizes, we can split lists into sublists of size at most p and deal with each sublist separately as before. Some care has to be taken, for example with the random choices of the bins, but details are mostly straightforward.

Algorithm 1 Layered 2-Choice Allocation (L2C)

L2C.Setup($\{(b_i, w_i)\}_{i=1}^n, w_{\max}$)

Receive n balls (b_i, w_i), and maximal total weight $w_{\max}$
Initialize $m = \lceil w_{\max}/(\delta(\lambda) \log\log w_{\max}) \rceil$ empty bins $B_1, ..., B_m$
for all $i \in \{1, ..., n\}$ **do**
    Set $\alpha_1, \alpha_2 \leftarrow \mathsf{H}(b_i)$
    InsertBall$(b_i, w_i, B_{\alpha_1}, B_{\alpha_2})$
Return $B_1, ..., B_m$

L2C.InsertBall($b_{\mathsf{new}}, w_{\mathsf{new}}, B_{\alpha_1}, B_{\alpha_2}$)

Receive bins $B_{\alpha_1}, B_{\alpha_2}$, and ball $(b_{\mathsf{new}}, w_{\mathsf{new}})$
Assert that α_1, α_2 are the choices given by $\mathsf{H}(b_{\mathsf{new}})$
Split the set of possible weights $[0, 1]_{\mathbb{R}}$ into $\log\log m$ sub-intervals

$$[0, 1/\log m]_{\mathbb{R}}, (1/\log m, 2/\log m]_{\mathbb{R}}, ..., (2^{\log\log m - 1}/\log m, 1]_{\mathbb{R}}$$

Choose $k \in \mathbb{N}$ minimal such that $w_{\mathsf{new}} \leq 2^k/\log m$
if k = 1 **then**
    Set $\alpha \leftarrow \alpha_1$
else
    Let B_α be the bin with the least number of balls of weight in $\left(\frac{2^{k-1}}{\log m}, \frac{2^k}{\log m}\right]_{\mathbb{R}}$ among B_{α_1} and B_{α_2}
Insert ball b_{new} into bin B_α

L2C.UpdateBall($b_{\mathsf{old}}, w_{\mathsf{old}}, w_{\mathsf{new}}, B_{\alpha_1}, B_{\alpha_2}$)

Receive bins $B_{\alpha_1}, B_{\alpha_2}$ that contain ball $(b_{\mathsf{old}}, w_{\mathsf{old}})$, and new weight $w_{\mathsf{new}} \geq w_{\mathsf{old}}$
Assert that α_1, α_2 are the choices given by $\mathsf{H}(b_{\mathsf{old}})$
if $w_{\mathsf{old}}, w_{\mathsf{new}} \in \left(\frac{2^{k-1}}{\log m}, \frac{2^k}{\log m}\right]_{\mathbb{R}}$ for some k **then**
    Update the weight of b_{old} to w_{new} directly
else
    Mark b_{old} as residual ball (it is still considered as a ball of weight w_{old})
    InsertBall$(b_{\mathsf{old}}, w_{\mathsf{new}}, B_{\alpha_1}, B_{\alpha_2})$

L2C.UpdateBall receives old ball (w, L), identifiers L' and bins $(B_{\alpha_1}, B_{\alpha_2})$, and updates ball (w, L) to ball $(w, L \cup L')$ as before, while merging both identifier lists L and L'. (The weight of the updated ball is $|L \cup L'|/p \in [0, 1]_{\mathbb{R}}$.)

5.1 LayeredSSE

We describe LayeredSSE, focusing on insert operations. In the full version, we describe LayeredSSE in more detail, and show how to treat arbitrary list sizes, introduce delete operations and show how to obtain updates in 1 RTT. A detailed description of LayeredSSE is given in algorithm 2.

Setup. To setup the initial database $\mathsf{DB} = (w, L_i)_{i=1}^W$, given upperbound N on the number of keyword-identifiers, allocate the balls (w, L_i) into m bins via L2C. Next, each bin is filled up to maximal size $p \cdot c \log\log\log(\lambda) \log\log(N/p)$, for some constant c. Finally, the encrypted bins are output.

Search. During a search operation on keyword w, the client retrieves encrypted bins $B_{\alpha_1}, B_{\alpha_2}$ for $(\alpha_1, \alpha_2) \leftarrow \mathsf{H}(w)$ from the server.

Update. During an update operation to add identifier list L' to keyword w, the client retrieves $B_{\alpha_1}, B_{\alpha_2}$, decrypts both bins and retrieves ball (w, L) from the corresponding bin $B_\alpha \in \{B_{\alpha_1}, B_{\alpha_2}\}$. Then, she calls $\mathsf{L2C.UpdateBall}$ with old ball (w, L), new identifiers L' and bins $B_{\alpha_1}, B_{\alpha_2}$ to insert the new identifiers L' into one of the bins. Finally, she reencrypts the bins and sends them to the server. The server then replaces the old bins with the updated bins.

Algorithm 2 LayeredSSE

Global parameters: constant $c \in \mathbb{N}$, page size p

$\mathsf{LayeredSSE.KeyGen}(1^\lambda)$

1: Sample $\mathsf{K}_{\mathsf{Enc}}$ for Enc with input 1^λ
2: **return** $K = \mathsf{K}_{\mathsf{Enc}}$

$\mathsf{LayeredSSE.Setup}(\mathsf{K}, N, \mathsf{DB})$

1: Set $\tau \leftarrow p \cdot c \log\log\log(\lambda) \log\log(N/p)$
2: Sample bins $B_1, \ldots, B_m$ via $\mathsf{L2C.Setup}$ with input $(\{(w_i, \mathsf{DB}(w_i))\}_{i=1}^W, N/p)$
3: Fill $B_1, \ldots, B_m$ up to size τ with zeros
4: Set $B_i^{\mathsf{enc}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_i)$ for $i \in [1, m]$
5: **return** $\mathsf{EDB} = (B_1^{\mathsf{enc}}, \ldots, B_m^{\mathsf{enc}})$

$\mathsf{LayeredSSE.Search}(\mathsf{K}, w; \mathsf{EDB})$

Client:

1: **return** w

Server:

1: Set $\alpha_1, \alpha_2 \leftarrow \mathsf{H}(w)$
2: **return** $B_{\alpha_1}^{\mathsf{enc}}, B_{\alpha_2}^{\mathsf{enc}}$

$\mathsf{LayeredSSE.Update}(\mathsf{K}, (w, L'), \mathsf{add}; \mathsf{EDB})$

Client:

1: **return** w

Server:

1: Set $\alpha_1, \alpha_2 \leftarrow \mathsf{H}(w)$
2: **return** $B_{\alpha_1}^{\mathsf{enc}}, B_{\alpha_2}^{\mathsf{enc}}$

Client:

1: Set $B_{\alpha_i} \leftarrow \mathsf{Dec}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\alpha_i}^{\mathsf{enc}})$ for $i \in \{1, 2\}$
2: Retrieve ball (w, L) from B_α for appropriate $\alpha \in \{\alpha_1, \alpha_2\}$
3: Run $\mathsf{L2C.UpdateBall}((w, L), L', B_{\alpha_1}, B_{\alpha_2})$
4: Set $B_{\alpha_i}^{\mathsf{new}} \leftarrow \mathsf{Enc}_{\mathsf{K}_{\mathsf{Enc}}}(B_{\alpha_i})$ for $i \in \{1, 2\}$
5: **return** $B_{\alpha_2}^{\mathsf{new}}, B_{\alpha_2}^{\mathsf{new}}$

Server:

1: Replace $B_{\alpha_i}^{\mathsf{enc}}$ with $B_{\alpha_i}^{\mathsf{new}}$ for $i \in \{1, 2\}$

5.2 Security and Efficiency

Correctness. $\mathsf{LayeredSSE}$ is correct as each keyword has two bins that contain its identifiers associated to it (and these bins are consistently retrieved and updated with $\mathsf{L2C}$). If the hash function is modeled as a random oracle, the bin choices are uniformly random and Theorem 1 guarantees that bins do not overflow.

Selective Security. $\mathsf{LayeredSSE}$ is selectively secure and has standard setup leakage N, such as search and update leakage qp, where qp is the query pattern[4]. This can be shown with a simple hybrid argument, sketched here. For setup, the simulator Sim receives N, recomputes m and initializes m empty bins $B_1, \ldots, B_m$ of size $p \cdot c \log\log\log(\lambda) \log\log(N/p)$ each. Sim then outputs $\mathsf{EDB}' = (\mathsf{Enc}_{\mathsf{K}'_{\mathsf{Enc}}}(B_i)_{i=1}^m)$ for some sampled key $\mathsf{K}'_{\mathsf{Enc}}$. As Enc is $\mathsf{IND\text{-}CPA}$ secure (and bins do not overflow in the real experiment except with negligible probability), the output EDB' is indistinguishable from the output of Setup in the real

[4] This is equivalent to page length hiding leakage $\mathcal{L}_{\mathsf{len\text{-}hid}}$, as we only restrict ourselves to lists of size at most p.

experiment. For a search query on keyword w, Sim checks the query pattern qp whether w was already queried. If w was not queried before, Sim a new uniformly random keyword w'. Otherwise, Sim responds with the same keyword w' from the previous query. As we assume that keywords are preprocessed by the client via a PRF, the keywords w and w' are indistinguishable. For an update query on keyword w, the client output in the first flow is the same as in a search query and thus, Sim can proceed as in search. For the second flow, Sim receives two bins $B_{\alpha_1}, B_{\alpha_2}$ from the adversary, directly reencrypts them and sends them back to the adversary. This behaviour is indistinguishable, as the bins are encrypted and again, bins do not overflow except with negligible probability.

Adaptive Security. For adaptive security, the adversary can issue search and update queries that depend on previous queries. As Theorem 1 assumes selectively chosen InsertBall and UpdateBall operations, there is no guarantee that bins do not overflow anymore in the real game. Thus, the adversary can potentially distinguish update queries of the simulated game from real update queries if she manages to overflow a bin in the real game, as she would receive bins with increased size only in latter case. Fortunately, we can just add a check in Update whether one of the bins overflows after the L2C.UpdateBall operation. In that case, the client reverts the update and send back the (reencrypted) original bins. Now, Theorem 1 still guarantees that bins overflow only with negligible probability after Setup and we can show that the simulated game is indistinguishable from the real game as before. Note that LayeredSSE is still correct after this modification, since updates that lead to overflows cannot occur by accident, but only if the client systemically adapts the choice of updates to the random coins used during previous update operations (see Theorem 1).

Note that when the client remarks that a bin overflowed in an Update in a real world environment, this is due malicious Update operations. The client can adapt his reaction accordingly, whereas the server learns no information about the attack without being notified by the client. We can show that LayeredSSE with the adjustment of Update is correct and $\mathcal{L}_{\mathsf{len\text{-}hid}}$-adaptively secure. The same simulator Sim suffices and we omit the details.

Efficiency. LayeredSSE has constant storage efficiency, as the server stores $m = \left\lceil (N/p)/(\log\log\log\lambda \cdot \log\log\frac{N}{p}) \right\rceil$ bins of $\mathcal{O}\left(p \log\log\log\lambda \cdot \log\log\frac{N}{p}\right)$ identifiers each. There is no client stash required. Each search and update query, the server looks up 2 bins, and thus LayeredSSE has $\widetilde{\mathcal{O}}\left(\log\log(N/p)\right)$ page efficiency. Note that LayeredSSE has $\mathcal{O}(1)$ locality if only lists up to size p are inserted.

Extensions. With some care, LayeredSSE can handle deletes and arbitrary lists (without sacrificing security and efficiency). We refer to the full version for more details. The results are formalized in Theorem 2.

Theorem 2 (LayeredSSE)**.** *Let N be an upper bound on the size of database* DB *and $p \leq N^{1-1/\log\log\lambda}$ be the page size. The scheme* LayeredSSE *is correct and $\mathcal{L}_{\mathsf{len\text{-}hid}}$-adaptively semantically secure if* Enc *is* IND-CPA *secure and* H *is modeled*

as a random oracle. It has constant storage efficiency and $\widetilde{\mathcal{O}}(\log\log N/p)$ *page efficiency. If only lists up to size* p *are inserted,* LayeredSSE *has constant locality.*

6 The Generic Local Transform

In this section, we define the Generic Local Transform (GLT), creating a link between the two IO-efficiency goals of *locality* and *page efficiency*. Namely, the GLT builds an SSE scheme with good *locality* properties from an SSE scheme with good *page efficiency*. For a page-efficient scheme to be used within the GLT, it needs to have certain extra properties. We define such schemes as *suitable* page-efficient schemes in Section 6.1. Next, we introduce the useful notion of *overflowing* SSE. The GLT is then obtained by combining an overflowing SSE with a suitable page-efficient scheme. The OSSE we will use for that purpose, ClipOSSE, is presented in Section 6.2. Finally, the GLT is built from the previous components in Section 6.4. An overview of the correctness and security proofs is provided in section 6.5. Full proofs are available in the full version.

6.1 Preliminaries

Suitable page-efficient SSE.

The GLT will create many instances of the underlying page-efficient scheme, each with a different page size. For that reason, for the purpose of the GLT, we slightly extend the standard SSE interface defined in Section 3: namely, $\mathsf{Setup}(\mathsf{K}, N, \mathsf{DB}, p)$ takes as an additional parameter the page size p. In addition, recall that, in Section 3, we have allowed the $\mathsf{Update}(\mathsf{K}, (w, L), \mathsf{op}, \mathsf{st}; \mathsf{EDB})$ procedure to add a *set* of matching documents K to a given keyword w in a single call. Note that S is allowed to be empty, in which case nothing is added.

If a scheme instantiates that interface, and, in addition, satisfies the following three conditions, we will call such as scheme a *suitable* page-efficient SSE.

- The scheme has client storage $\mathcal{O}(1)$.
- The scheme has locality $\mathcal{O}(1)$ during searches and updates *when accessing a list of length at most one page.*
- The leakage of the scheme is page-length-hiding.

Overflowing SSE. We introduce the notion of *Overflowing* SSE. An Overflowing SSE (OSSE) has the same interface and functionality as a standard SSE scheme, except that during a Setup or Update operation, it may refuse to store some document identifiers. Those identifiers are called *overflowing*. At the output of the Setup and Update operations, the client returns the set of overflowing elements. Compared to standard SSE, the correctness definition is relaxed in the following way: during a Search, only matching identifiers that were *not* overflowing need to be retrieved.

The intention of an Overflowing SSE is that it may be used as a component within a larger SSE scheme, which will store the overflowing identifiers using

a separate mechanism. The use of an OSSE may be regarded as implicit in some prior SSE constructions. We have chosen to introduce the notion explicitly because it allows to cleanly split the presentation of the Generic Local Transform into two parts: an OSSE scheme that stores most of the database, and an array of page-efficient schemes that store the overflowing identifiers.

6.2 Dynamic Two-Dimensional One-Choice Allocation

The first component of the Generic Local Transform is an OSSE scheme, ClipOSSE. In line with prior work, we split the presentation of ClipOSSE into two parts: an allocation scheme, which specifies where elements should be stored; and the SSE scheme built on top of it, which adds a layer of encryption, key management, and other mechanisms needed to convert the allocation scheme into a full SSE.

The allocation scheme within ClipOSSE is called 1C-Alloc. Similar to [ANSS16], the allocation scheme is an abstract construct that defines the memory locations where items should be stored, but does not store anything itself. In the case of 1C-Alloc, items are stored within buckets, and the procedures return as output the indices of *buckets* where items should be stored. From the point of view of 1C-Alloc, each bucket has unlimited storage. In more detail, 1C-Alloc contains two procedures, Fetch and Add.

- $\mathsf{Fetch}(m, w, \ell)$: given a number of buckets m, a keyword w, and a list length ℓ, Fetch returns (a superset of) the indices of buckets where elements matching keyword w may be stored, assuming there are ℓ such elements.
- $\mathsf{Add}(m, w, \ell)$: given the same input, Add returns the index of the bucket where the *next* element matching keyword w should be inserted, assuming there are currently ℓ matching elements.

The intention is that Add is used during an SSE Update operation, in order to choose the bucket where the next list element is stored; while Fetch is used during a Search operation, in order to determine the buckets that need to be read to retrieve all list elements. 1C-Alloc will satisfy the correctness property given in Definition 10. Note that the number of buckets m is always assumed to be a power of 2.

Definition 10 (Correctness). *For all m, w, ℓ, if m is a power of 2, then*

$$\bigcup_{0 \le i \le \ell - 1} \mathsf{Add}(m, w, i) \subseteq \mathsf{Fetch}(m, w, \ell).$$

To describe 1C-Alloc, it is convenient to conceptually group buckets into superbuckets. For $\ell = 2^i \le m$, an *ℓ-superbucket* is a collection of ℓ consecutive buckets, with indices of the form $k \cdot \ell, k \cdot \ell + 1, \ldots, (k+1) \cdot \ell - 1$, for some $k \le m/\ell$. A 1-superbucket is the same as a bucket. Notice that for a given ℓ, the ℓ-superbuckets do not overlap. They form a partition of the set of buckets. For $\ell > 1$, each ℓ-superbucket contains exactly two $\ell/2$-superbuckets.

Let H be a hash function, whose output is assumed to be uniformly random in $\{1, \ldots, m\}$. 1C-Alloc works as follows. Fix a keyword w and length $\ell \le m$

(the case $\ell > m$ will be discussed later). Let $\ell' = 2^{\lceil \log \ell \rceil}$ be the smallest power of 2 larger than ℓ. On input w and ℓ, 1C-Alloc.Fetch returns the (unique) ℓ'-superbucket that contains $H(w)$.

Algorithm 3 Dynamic Two-Dimensional One-Choice Allocation (1C-Alloc)

1C-Alloc.Fetch(m, w, ℓ)

1: $\ell' \leftarrow 2^{\lceil \log \ell \rceil}$
2: **if** $\ell' \geq m$ **then**
3: **return** $\{0, \ldots, m-1\}$
4: **else**
5: $i \leftarrow \lfloor H(w)/\ell' \rfloor$
6: **return** $\{\ell' \cdot i, \ldots, \ell' \cdot i + \ell' - 1\}$

1C-Alloc.Add(m, w, ℓ)

1: $\ell \leftarrow \ell \bmod m$
2: $\ell' \leftarrow 2^{\lceil \log(\ell+1) \rceil}$
3: $i \leftarrow \lfloor H(w)/\ell' \rfloor$
4: **if** $\lfloor 2H(w)/\ell' \rfloor \bmod 2 = 0$ **then**
5: **return** $\ell' \cdot i + \ell$
6: **else**
7: **return** $\ell' \cdot i + \ell - \ell'/2$

Meanwhile, 1C-Alloc.Add is designed in order to ensure that the first ℓ successive locations returned by Add for keyword w are in fact included within the ℓ'-superbucket above $H(w)$ (that is, in order to ensure correctness). For the first list element (when $\ell = 0$), Add returns the bucket $H(w)$; for the second element, it returns the other bucket contained inside the 2-superbucket above $H(w)$. More generally, if S is the smallest superbucket above $H(w)$ that contains at least $\ell+1$ buckets, Add returns the leftmost bucket within S that has not yet received an element. In practice, the index of that bucket can be computed easily based on ℓ and the binary decomposition of $H(w)$, as done in Algorithm 3. (In fact, the exact order in which buckets are selected by Add is irrelevant, as long as it selects distinct buckets, and correctness holds.)

When the size of the list ℓ grows above the number of buckets m, Fetch returns all buckets, while Add selects the same buckets as it did for $\ell \bmod m$.

6.3 Clipped One-Choice OSSE

ClipOSSE is the OSSE scheme obtained by storing lists according to 1C-Alloc, using $m = O(N/\log\log N)$ buckets, with each bucket containing up to $\tau = \lceil \alpha \log\log N \rceil$ items, for some constant α. Buckets are always padded to the threshold τ and encrypted before being stored on the server. Thus, from the server's point of view, they are completely opaque. A table T containing (in encrypted form) the length of the list matching each keyword w is also stored on the server.

Given 1C-Alloc, the details of ClipOSSE are straightforward. A short overview is given in text below. The encrypted database generated by Setup is essentially equivalent to starting from an empty database, and populating it by making repeated calls to Update, one for each keyword–document pair in the database. For that reason, we focus on Search and Update. The full specification for Setup, Search, and Update is given as pseudo-code in Algorithm 4.

Search. To retrieve the list of identifiers matching keyword w, ClipOSSE calls 1C-Alloc(m, w, ℓ) to get the set of bucket indices where the elements matching keyword w have been stored. The client retrieves those buckets from the server, and decrypts them to obtain the desired information.

Update. For simplicity, we focus on a the case where a single identifier is added. The case of a set of identifiers can be obtained by repeating the process for each identifier in the set. To add the new item to the list matching keyword w, ClipOSSE calls 1C-Alloc(m, w, ℓ) to determine the bucket where the new list item should be inserted. The client retrieves that bucket from the server, decrypts it, adds the new item, reencrypts the bucket, and sends it back to the server. If that bucket was already full, the item is overflowing, in the sense of Section 6.1.

Algorithm 4 Clipped One-Choice OSSE (ClipOSSE)

Global parameters: constants $d, \alpha \in \mathbb{N}^*$

ClipOSSE.KeyGen(1^λ)

Generate keys K, K_{PRF} for Enc, PRF
return $\mathsf{K} = (K, K_{\mathsf{PRF}})$

ClipOSSE.Setup$(\mathsf{K}, N, \mathsf{DB})$

$m \leftarrow 2^{\lceil \log(N/\log\log N) \rceil}$
$\tau \leftarrow \lceil \alpha \log\log N \rceil$
$B_0, \ldots, B_{m-1}, T, \mathsf{EDB}, \mathsf{clip} \leftarrow \varnothing$
for all each $(w, \{e_1, \ldots, e_\ell\})$ in DB **do**
  $K_w \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w)$
  $T[w] \leftarrow \mathsf{Enc}_{K_w}(\ell)$
  for all t from 1 to ℓ **do**
    $C \leftarrow \varnothing$
    $i \leftarrow$ 1C-Alloc.Add$(m, w, t-1)$
    if then$|B[i]| < \tau$
      $B[i] \leftarrow B[i] \cup \{e_i\}$
    else
      $C \leftarrow C \cup \{e_i\}$
    if $|S| > 0$ **then**
      $\mathsf{clip} \leftarrow \mathsf{clip} \cup (w, \ell, C)$
Let $B^{\mathsf{Enc}}[i] = \mathsf{Enc}_K(B_i)$ for each i
return $\mathsf{EDB} = (T, (B^{\mathsf{Enc}}[i])), \mathsf{clip}$

ClipOSSE.Search$(\mathsf{K}, w, \mathsf{st}; \mathsf{EDB})$

Client: *(search token)*
send $(w, K_w) = \mathsf{PRF}_{K_{\mathsf{PRF}}}(w)$
Server:
$\ell \leftarrow \mathsf{Dec}_{K_w}(T[w])$
$S \leftarrow$ 1C-Alloc.Fetch(m, w, ℓ)
return $\{B^{\mathsf{Enc}}[i] : i \in S\}$

ClipOSSE.Update$(\mathsf{K}, (w, \{e\}), \mathsf{op}, \mathsf{st}; \mathsf{EDB})$

Client: *(update token)*
send $(w, K_w = \mathsf{PRF}_{K_{\mathsf{PRF}}}(w))$
Server:
$\ell \leftarrow \mathsf{Dec}_{K_w}(T[w])$
$i \leftarrow$ 1C-Alloc.Add(m, w, ℓ)
send $B^{\mathsf{Enc}}[i]$
Client:
$B \leftarrow \mathsf{Dec}_K(B^{\mathsf{Enc}}[i])$
if $|B| < \tau$ **then**
  $\mathsf{clip} \leftarrow \varnothing$
  $B \leftarrow B \cup \{e\}$
else
  $\mathsf{clip} \leftarrow \{e\}$
send $B' = \mathsf{Enc}_K(B)$
Server:
$B^{\mathsf{Enc}}[i] \leftarrow B'$
Client:
return clip

6.4 The Generic Local Transform

The Generic Local Transform takes as input a page-length-hiding page-efficient SSE scheme PE-SSE. It outputs a local SSE scheme Local[PE-SSE].

To realize Local[PE-SSE], we use two structures. The first structure is an instance of ClipOSSE, which stores most of the database. The second structure is an array of n_{level} instances of PE-SSE. The i-th instance, denoted PE-SSE$_i$, has page size 2^i. The PE-SSE$_i$ instances are used to store elements that overflow

from ClipOSSE. In addition, a table T stores (in encrypted form) the length of the list matching keyword w, for each keyword[5].

Fix a keyword w, matching ℓ elements. Let $\ell' = 2^{\lceil \log \ell \rceil}$ be the smallest power of 2 larger than ℓ. Let $i = \log \ell'$. At any point in time, the elements matching w are stored in two locations: ClipOSSE, and PE-SSE$_i$. Each of these two locations stores part of the elements: ClipOSSE stores the elements that did not overflow, and PE-SSE$_i$ stores the overflowing elements. Each element exists in only one of the two locations. Again, for simplicity, we define updates for adding a single identifier per keyword. The case of adding a set of identifiers at once can be deduced by repeating the same process for each identifier in the set.

Algorithm 5 Generic Local Transform (Local[PE-SSE])

Global parameters: constant $d \in \mathbb{N}^*$

Local[PE-SSE].KeyGen(1^λ)

1: Generate key K_{PRF} for PRF
2: **return** $\mathsf{K} = (K, K_{\mathsf{PRF}})$

Local[PE-SSE].Update(K, (w, L); EDB)

Client: *(update token)*

1: **send** $(w, L, K_w = \mathsf{PRF}_{K_{\mathsf{PRF}}}(w))$

Server:

1: $C \leftarrow$ ClipOSSE.Update(w, L)
2: $\ell \leftarrow \mathsf{Dec}_{K_w}(T[w])$
3: $T[w] \leftarrow \mathsf{Enc}_{K_w}(\ell + 1)$
4: **send** ℓ

Client:

1: $i \leftarrow \lceil \log \ell \rceil$
2: **if** $\lceil \log \ell \rceil = \lceil \log(\ell + 1) \rceil$ **then**
3: PE-SSE$_i$.Update(w, C)
4: **else**
5: $S \leftarrow$ set of matches in PE-SSE$_i$.Search(w)
6: PE-SSE$_{i+1}$.Update$(w, S \cup C)$

Local[PE-SSE].Setup(K, N, DB)

1: $n_{level} \leftarrow \lceil N / \log^d N \rceil$
2: **for all** $(w, S) \in$ DB **do**
3: $K_w \leftarrow \mathsf{PRF}_{K_{\mathsf{PRF}}}(w)$
4: $T[w] \leftarrow \mathsf{Enc}_{K_w}(|S|)$
5: EDB, clip $\leftarrow$ ClipOSSE.Setup(DB)
6: **for all** i from 0 to n_{level} **do**
7: $\mathsf{DB}_i \leftarrow \{(w, C) : (w, \ell, C) \in \mathsf{clip}$ and $2^{i-1} < \ell \leq 2^i\}$
8: PE-SSE$_i \leftarrow$ PE-SSE.Setup$(\lceil N / \log N \rceil, 2^i, \mathsf{DB}_i)$

Local[PE-SSE].Search(K, w, st; EDB)

Client: *(search token)*

1: **send** $(w, K_w = \mathsf{PRF}_{K_{\mathsf{PRF}}}(w))$

Server:

1: $i \leftarrow \lceil \log(\mathsf{Dec}_{K_w}(T[w])) \rceil$
2: **return** ClipOSSE.Search(w) $\cup$ PE-SSE$_i$.Search(w)

Search. During a search operation, Local[PE-SSE] queries both structures, and combines their output to retrieve all matching elements.

Update. During an update operation to add element e, Local[PE-SSE] forwards the update query to ClipOSSE, and gets as output $C = \varnothing$ if the element did not overflow, or $C = \{e\}$ if the element did overflow. For now, assume that $\lceil \log \ell \rceil = \lceil \log(\ell + 1) \rceil$, that is, the PE-SSE$_i$ instance associated with the list remains the same during the update operation. In that case, PE-SSE$_i$ is updated for the set C. (Recall from Section 6.1 that a length-hiding SSE such as PE-SSE accepts sets of elements as input in Update.) The length-hiding property is designed to guarantee that the content of C (including whether it is empty) is not

[5] The same table exists in ClipOSSE. In an actual implementation, they would be the same table, but using ClipOSSE in black box eases the presentation.

leaked to the server. Now assume $\lceil \log \ell \rceil < \lceil \log(\ell+1) \rceil$. In that case, the PE-SSE instance associated with the list becomes $\mathsf{PE\text{-}SSE}_{i+1}$ instead of $\mathsf{PE\text{-}SSE}_i$. The client retrieves all current overflowing elements from $\mathsf{PE\text{-}SSE}_i$, adds the content of C, and stores the result in $\mathsf{PE\text{-}SSE}_{i+1}$.

6.5 Overflow of ClipOSSE

The main technical result in this section regards the number of overflowing items in ClipOSSE.

Theorem 3. *Suppose that* ClipOSSE *receives as input a database of size N, such that the size of the longest list is $\mathcal{O}\left(N/\log^d N\right)$ for some $d \geq 2$. Then for any constant c, there exists a choice of parameters of* ClipOSSE *such that the number of overflowing items is $\mathcal{O}(N/\log^c N)$.*

The proof of Theorem 3 is intricate. For space reasons, we only give a brief overview here. A detailed overview and the full proof is given in the full version.. First, we show that the result holds in the special case where all lists have length $N/\log^d N$. This uses a negative association argument, similar to the proof of [DPP18, Theorem 1]. The core of the proof is to then show that this special case implies the general case. This is done by iteratively merging short lists, while showing that this merging process can only have a limited effect on the number of overflowing elements. At the outcome of the merging process, all lists have length $N/\log^d N$, which reduces the problem to the special case. The main technique for the reduction is a stochastic dominance argument, combined with a convexity argument (similar to the proof of [BBF+21, Theorem 5]).

The Generic Local Transform itself uses standard SSE techniques, and its properties follow from previous discussions. We provide a formal statement below.

Theorem 4 (Generic Local Transform). *Let N be an upper bound on the size of database* DB*. Suppose that* PE-SSE *is a* suitable *page-efficient scheme with page efficiency P and storage efficiency S. Then* Local[PE-SSE] *is a correct and secure SSE scheme with storage efficiency $\mathcal{O}(S)$, locality $\mathcal{O}(1)$, and read efficiency $P + \widetilde{\mathcal{O}}(\log \log N)$.*

References

ABKU94. Azar, Y., Broder, A.Z., Karlin, A.R., and Upfal, E. Balanced allocations. In: Proceedings of the twenty-sixth annual ACM symposium on theory of computing, pp. 593–602 (1994).

AKM19. Amjad, G., Kamara, S., and Moataz, T. Breach-resistant structured encryption. Proceedings on Privacy Enhancing Technologies, vol. 2019(1):(2019), pp. 245–265.

ANSS16. Asharov, G., Naor, M., Segev, G., and Shahaf, I. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: D. Wichs and Y. Mansour (eds.), 48th Annual ACM Symposium on Theory of Computing, pp. 1101–1114. ACM Press, Cambridge, MA, USA (Jun. 18–21, 2016).

ASS18. Asharov, G., Segev, G., and Shahaf, I. Tight tradeoffs in searchable symmetric encryption. In: H. Shacham and A. Boldyreva (eds.), Advances in Cryptology – CRYPTO 2018, Part I, *Lecture Notes in Computer Science*, vol. 10991, pp. 407–436. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug. 19–23, 2018).

ASS21. Asharov, G., Segev, G., and Shahaf, I. Tight tradeoffs in searchable symmetric encryption. Journal of Cryptology, vol. 34(2):(2021), pp. 1–37.

BBF+21. Bossuat, A., Bost, R., Fouque, P.A., Minaud, B., and Reichle, M. SSE and SSD: Page-efficient searchable symmetric encryption. In: T. Malkin and C. Peikert (eds.), Advances in Cryptology – CRYPTO 2021, Part III, *Lecture Notes in Computer Science*, vol. 12827, pp. 157–184. Springer, Heidelberg, Germany, Virtual Event (Aug. 16–20, 2021).

BFHM08. Berenbrink, P., Friedetzky, T., Hu, Z., and Martin, R. On weighted balls-into-bins games. Theoretical Computer Science, vol. 409(3):(2008), pp. 511–520.

BMO17. Bost, R., Minaud, B., and Ohrimenko, O. Forward and backward private searchable encryption from constrained cryptographic primitives. In: B.M. Thuraisingham, D. Evans, T. Malkin, and D. Xu (eds.), ACM CCS 2017: 24th Conference on Computer and Communications Security, pp. 1465–1482. ACM Press, Dallas, TX, USA (Oct. 31 – Nov. 2, 2017).

Bos16. Bost, R. $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, and S. Halevi (eds.), ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 1143–1154. ACM Press, Vienna, Austria (Oct. 24–28, 2016).

CGKO06. Curtmola, R., Garay, J.A., Kamara, S., and Ostrovsky, R. Searchable symmetric encryption: improved definitions and efficient constructions. In: A. Juels, R.N. Wright, and S. De Capitani di Vimercati (eds.), ACM CCS 2006: 13th Conference on Computer and Communications Security, pp. 79–88. ACM Press, Alexandria, Virginia, USA (Oct. 30 – Nov. 3, 2006).

CJJ+14. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In: ISOC Network and Distributed System Security Symposium – NDSS 2014. The Internet Society, San Diego, CA, USA (Feb. 23–26, 2014).

CK10. Chase, M. and Kamara, S. Structured encryption and controlled disclosure. In: M. Abe (ed.), Advances in Cryptology – ASIACRYPT 2010, *Lecture Notes in Computer Science*, vol. 6477, pp. 577–594. Springer, Heidelberg, Germany, Singapore (Dec. 5–9, 2010).

CT14. Cash, D. and Tessaro, S. The locality of searchable symmetric encryption. In: P.Q. Nguyen and E. Oswald (eds.), Advances in Cryptology – EUROCRYPT 2014, *Lecture Notes in Computer Science*, vol. 8441, pp. 351–368. Springer, Heidelberg, Germany, Copenhagen, Denmark (May 11–15, 2014).

DP17. Demertzis, I. and Papamanthou, C. Fast searchable encryption with tunable locality. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1053–1067 (2017).

DPP18. Demertzis, I., Papadopoulos, D., and Papamanthou, C. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In: H. Shacham and A. Boldyreva (eds.), Advances in Cryptology – CRYPTO 2018, Part I, *Lecture Notes in Computer Science*, vol. 10991, pp. 371–406. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug. 19–23, 2018).

EKPE18. Etemad, M., Küpçü, A., Papamanthou, C., and Evans, D. Efficient dynamic searchable encryption with forward privacy. Proceedings on Privacy Enhancing Technologies, vol. 2018(1):(2018), pp. 5–20.

JK77. Johnson, N.L. and Kotz, S. Urn models and their application; an approach to modern discrete probability theory. New York, NY (USA) Wiley (1977).

MM17. Miers, I. and Mohassel, P. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In: ISOC Network and Distributed System Security Symposium – NDSS 2017. The Internet Society, San Diego, CA, USA (Feb. 26 – Mar. 1, 2017).

MPC+18. Mishra, P., Poddar, R., Chen, J., Chiesa, A., and Popa, R.A. Oblix: An efficient oblivious search index. In: 2018 IEEE Symposium on Security and Privacy, pp. 279–296. IEEE Computer Society Press, San Francisco, CA, USA (May 21–23, 2018).

PR04. Pagh, R. and Rodler, F.F. Cuckoo hashing. Journal of Algorithms, vol. 51(2):(2004), pp. 122–144.

RMS01. Richa, A.W., Mitzenmacher, M., and Sitaraman, R. The power of two random choices: A survey of techniques and results. Combinatorial Optimization, vol. 9:(2001), pp. 255–304.

TW07. Talwar, K. and Wieder, U. Balanced allocations: the weighted case. In: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, pp. 256–265 (2007).

TW14. Talwar, K. and Wieder, U. Balanced allocations: A simple proof for the heavily loaded case. In: International Colloquium on Automata, Languages, and Programming, pp. 979–990. Springer (2014).

ZKP16. Zhang, Y., Katz, J., and Papamanthou, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: T. Holz and S. Savage (eds.), USENIX Security 2016: 25th USENIX Security Symposium, pp. 707–720. USENIX Association, Austin, TX, USA (Aug. 10–12, 2016).